

INSTITUT FÜR INFORMATIK  
UND PRAKTISCHE MATHEMATIK

**Removing Cycles in Esterel Programs**

Jan Lukoschus, Reinhard von Hanxleden

Bericht Nr. 0502

März 2005



CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der  
Christian-Albrechts-Universität zu Kiel  
Olshausenstr. 40  
D – 24098 Kiel

## **Removing Cycles in Esterel Programs**

Jan Lukoschus, Reinhard von Hanxleden

Bericht Nr. 0502  
März 2005

e-mail:  
jlu@informatik.uni-kiel.de,  
rvh@informatik.uni-kiel.de

An abbreviated version of this report will appear in the Proceedings of Synchronous Languages Applications and Programming (SLAP 2005), Edinburgh, April 3rd 2005 in Electronic Notes in Theoretical Computer Science (ENTCS).

## Abstract

Synchronous programs may contain cyclic signal interdependencies. This prohibits a static scheduling, which limits the choice of available compilation techniques for such programs. An earlier report [16] has outlined an algorithm which, given a constructive synchronous program, performs a semantics-preserving source-level code transformation that removes cyclic signal dependencies, and also exposes opportunities for further optimization. This report presents a refined algorithm, describes a method for computing the necessary replacement expressions, and presents experimental results obtained with a prototypical implementation of this algorithm using the Columbia Esterel Compiler.

# 1 Introduction

One of the strengths of synchronous languages [1] is their deterministic semantics in the presence of concurrency. It is possible to write a synchronous program which contains cyclic interdependencies among concurrent threads. Depending on the nature of this cycle, the program may still be valid; however, translating such a cyclic program poses challenges to the compiler. Therefore, not all approaches that have been proposed for compiling synchronous programs are applicable to cyclic programs. Hence, cyclic programs are currently only translatable by techniques that are relatively inefficient with respect to execution time, code size, or both. This paper proposes a technique for transforming valid, cyclic synchronous programs into equivalent acyclic programs, at the source-code level, thus extending the range of efficient compilation schemes that can be applied to these programs.

The focus of this paper is on the synchronous language Esterel [4]; however, the concepts introduced here should be applicable to other synchronous languages as well, such as Lustre [15].

Next we will provide a classification of cyclic programs, followed by an overview of previous work on compiling Esterel programs and handling cycles. Section 2 introduces the transformation algorithm for *pure signals*, which do not carry a value. Section 3 describes how to derive replacement expression for signals from an Esterel program. The application of the transformation algorithm is exemplified on two different types of cyclic dependencies in Section 4. Optimization options are presented in Section 5. Section 6 demonstrates the transformation with further examples, including a program using *valued signals*. Section 7 provides experimental results, the paper concludes in Section 8.

## 1.1 Cyclic Programs

The execution of an Esterel program is divided into discrete *instants*. An Esterel program communicates through *signals* that are either present or absent throughout each instant; this property is also referred to as the *synchrony hypothesis*. If a signal  $S$  is *emitted* in one instant, it is considered *present* from the beginning of that instant on. If a signal is not emitted in one instant, it is considered *absent*.

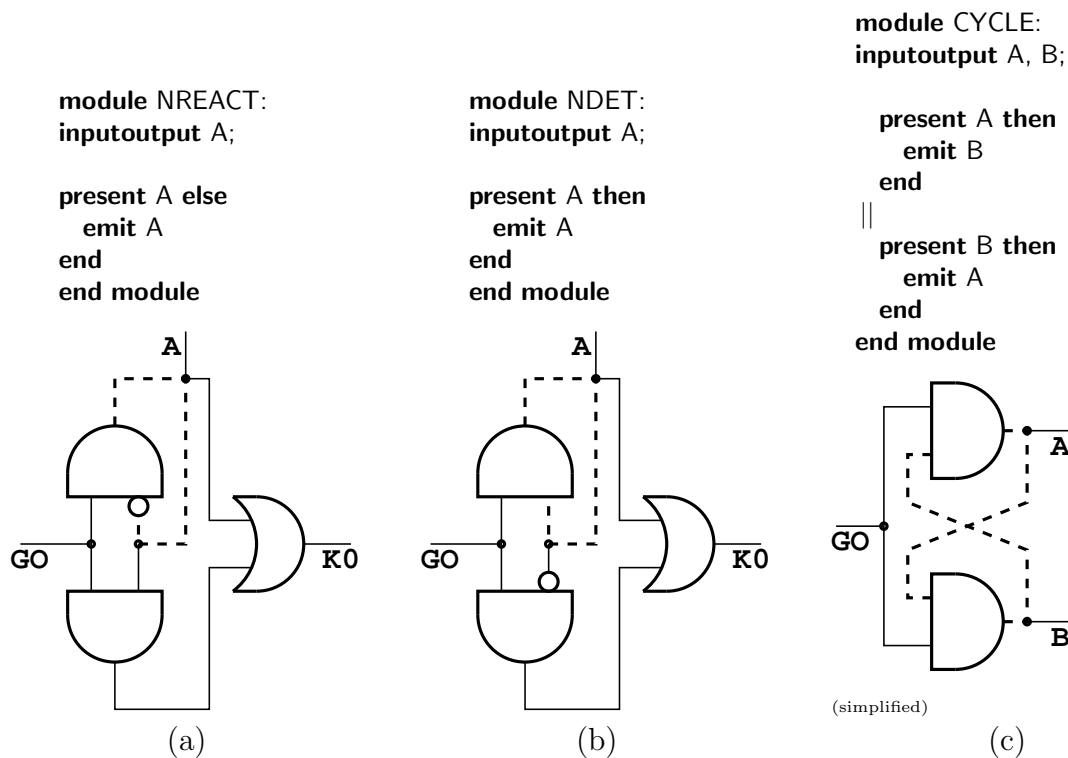


Figure 1: Invalid cyclic Esterel programs. The wires shown as dashed lines indicate the cyclic dependencies.

The Esterel language consists of a set of *primitive* statements, from which other statements are *derived* [2]. The primitives that directly involve signals are **signal** (signal declaration), **emit** (signal emission), **present** (conditional), and **suspend** (suspension).

The emission of signals can be conditionally executed depending on tests for the presence of other signals. This establishes *dependency relations* between signals. A closed circle of such dependency relations in an Esterel program is called a *dependency cycle*. Such a cycle is problematic, because the evaluation of a condition must not be invalidated by following signal emissions. If that occurs the program is invalid and must be rejected.

A simple example for such an invalid program is:

```

present A else
  emit A
end

```

When A is not present at the signal test then the **else** part is executed and A is emitted which invalidates the former signal test.

Consider the three short Esterel programs shown in Figure 1. The first program NREACT involves the signal A, which is an input signal, meaning that it can be emitted in the environment, and also an output signal, meaning that it can be tested by the environment. Here the environment may be either the external environment of the program, or it may consist of other Esterel modules. The body of NREACT states that

if  $A$  is present (emitted by the environment), then nothing is done, which is not problematic. However, if  $A$  is absent, then the `else` part is activated:  $A$  is emitted, which invalidates the former presence test for  $A$ . Such a contradiction is an invalid behavior of an Esterel program; such a program is over-constrained, or *not reactive*, and should be rejected by the compiler. This problem also becomes apparent when synthesizing this program into hardware, as the gate representation of this program is an inverter with its output directly fed back to the input. This is obviously not a stable circuit and hence forbidden in Esterel.

The program `NDET` in Figure 1(b) is similar to `NREACT`, but with `else` changed to `then`. Here a present  $A$  will result in an emission of  $A$  in the `then` branch of the `present` statement, which would justify taking the `then` branch. Conversely, an absent  $A$  will skip the emission of  $A$ . Hence, this program is under-constrained, or *not deterministic*. A compiler should reject `NDET`. This also becomes apparent at the gate representation of `NDET`, which is a driver gate that transmits the input value to the output. Now the output is fed back to the input to map the behavior of the program. As a certain gate delay is inevitable, this circuit may be an oscillator instead of providing stable outputs.

Programs `NREACT` and `NDET` have the same underlying problem: They involve a signal that is *self dependent*. In both programs the emission of  $A$  depends on a guard containing  $A$ . In these two examples, we have a *direct* self dependence, where the emission of a signal immediately depends on the presence of a signal. However, we may also have *indirect* self dependencies, in which a signal depends on itself via some other, intermediate signals. Consider program `CYCLE` in Figure 1(c), which contains two parallel threads, both testing for the signal emitted by the other one. However, the signals are emitted only if the other one has been emitted already; the emission of  $A$  depends on the presence of  $B$  and vice versa. In this case, we have a *cyclic dependency*, or *cycle* for short, and the program should again be rejected. We will refer to the emission of a signal that is guarded by a signal test (using `present`, `suspend`, or a derived statement) as a *guarded emit*.

All three programs shown in Figure 1 involve cyclic signal dependencies and are invalid, and hence of no further interest to us.

However, there are programs that contain dependency cycles and yet are valid. A program is considered valid, or *constructive*, if we can establish the presence or absence of each signal without speculative reasoning, which may be possible even if the program contains cycles. The equivalent formulation in hardware is that there are circuits that contains cycles and yet are self-stabilizing, irrespective of delays [3].

Consider the program `PAUSE_CYC` in Figure 2(a): the cyclic dependency consists of an emission of  $B$  guarded by a test for  $A$  and an emission of  $A$  guarded by a test for  $B$ . At run time, however, the dependencies are separated by a `pause` statement into separate execution instants. The emission of  $B$  in the first instant has no effect on the test for  $B$  in the second instance.

In such a case, where not all dependencies are active in the same execution instant, we will call the cyclic dependency a *false cycle*. In contrast, the programs shown in Figure 1 all contained *true cycles*, where all dependencies involved were present at the same instant.

```

module PAUSE_CYC:
input A, B;
output C;

```

```

    present A then
        emit B
    end;
    pause;
    present B then
        emit A
    end
||
    present B then
        emit C
    end
end module

```

(a)

```

module PAUSE_ACYC:
input A, B;
output C;
signal A_, B_, ST_0, ST_1, ST_2 in
    emit ST_0;
    [
        present [A or
            (ST_1 and (B or ST_0))] then
            emit B_
        end;
        pause; emit ST_1;
        present [B or B_] then
            emit A_
        end
    ||
        present [B or B_] then
            emit C
        end
    ] ; emit ST_2
end signal
end module

```

(c)

```

module PAUSE_PREP:
input A, B;
output C;
signal A_, B_, ST_0, ST_1, ST_2 in
    emit ST_0;
    [
        present [A or A_] then
            emit B_
        end;
        pause; emit ST_1;
        present [B or B_] then
            emit A_
        end
    ||
        present [B or B_] then
            emit C
        end
    ] ; emit ST_2
end signal
end module

```

(b)

```

module PAUSE_OPT:
input A, B;
output C;

signal B_ in
    present A then
        emit B_
    end;
    pause;
    present B then
        emit A
    end
||
    present [B or B_] then
        emit C
    end
end signal
end module

```

(d)

Figure 2: Resolving a false cycle.

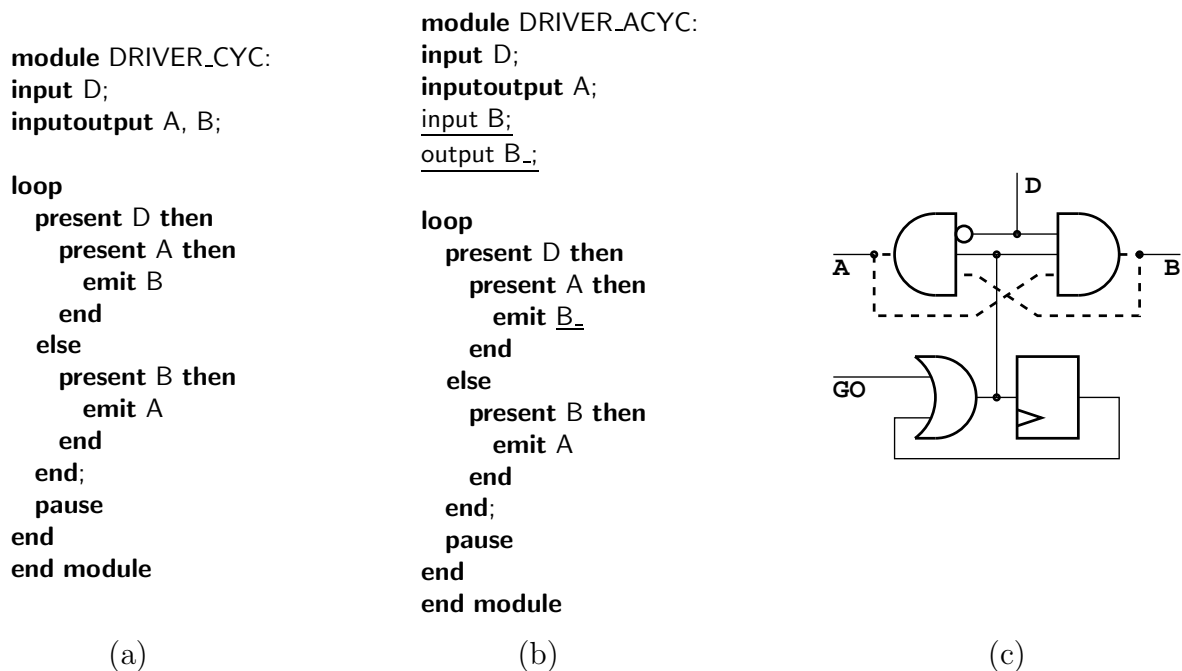


Figure 3: False cyclic dependencies in a bidirectional bus driver. The wires shown as dashed lines indicate the cyclic dependency.

A cycle may be false because it is broken by a register, as is the case in `PAUSE_CYC`, or because it is broken by a guard, as is the case in program `DRIVER_CYC` shown in Figure 3(a). Programs that only contain false cycles are still constructive and hence are valid programs that should be accepted by a compiler.

So far, we have considered only programs that contained true cycles and were invalid (`NREACT`, `NDET`, `CYCLE`) or that contained false cycles and were valid (`PAUSE_CYC`, `DRIVER_CYC`). However, there also exist programs that contain true cycles, with all dependencies evaluated at the same instant, and yet are valid programs. A classic example of a truly cyclic, yet constructive program is the Token Ring Arbiter [18]; Figure 4 shows a version with three stations. Each network station consists of two parallel threads: one computes the arbitration signals, the other passes a single token in each instant from one station to the next in each instant.

An inspection of the Arbiter reveals that there is a true cycle involving signals `P1`, `P2`, and `P3`. However, the program is still constructive as there is always at least one token present that breaks the cycle. Hence, a compiler should accept this program. Note that the same program, but without the first thread that emits `T1` in the first instant, should be rejected—this illustrates that determining constructiveness of a program is a non-trivial process.

## 1.2 Related Work

A number of different approaches for compiling Esterel programs into either software or hardware have been proposed.

```

module TR3_CYC:

input  R1, R2, R3;
output G1, G2, G3;

signal P1, P2, P3,
        T1, T2, T3
in
    emit T1
    ||
    loop % STATION1
        present [T1 or P1]
        then
            present R1 then
                emit G1
            else
                emit P2
            end
        end ;
        pause
    end loop
    ||

    loop
        present T1 then
            pause;
            emit T2
        else
            pause
        end
    end
    ||
    loop % STATION2
        present [T2 or P2]
        then
            present R2 then
                emit G2
            else
                emit P3
            end
        end ;
        pause
    end loop
    ||
    loop
        present T2 then
            pause;
            emit T3
        else
            pause
        end
    end
    ||
    loop % STATION3
        present [T3 or P3]
        then
            present R3 then
                emit G3
            else
                emit P1
            end
        end ;
        pause
    end loop
    ||
    loop
        present T3 then
            pause;
            emit T1
        else
            pause
        end
    end
end module

```

Figure 4: Token Ring Arbiter with three stations.



The approach used by the v5 compiler [5] is to translate an Esterel program into a net-list, which can either be realized in hardware or which can be simulated in software. Using the technique proposed by Shiple *et al.* [20], this approach handles cycles by re-synthesizing cyclic portions into acyclic portions, employing the algorithm by Bourdoncle [6]. However, the software variant tends to be rather slow, as it simulates the complete circuit during each instant, irrespective of which parts of the circuit are currently active.

Another approach implemented in the v5 compiler builds an automaton through exhaustive simulation of the netlist code. The resulting code is very fast, but potentially very large, as it is affected by possible state explosion.

A third approach to synthesize software is to generate an event-driven simulator, which breaks the simulated circuit into a number of small functions that are conditionally executed [8, 10, 7]. These compilers tend to produce code that is compact and yet almost as fast as automata-based code. The drawback of these techniques is that so far, they rely on the existence of a static schedule and hence are limited to acyclic programs. One approach to overcome this limitation, which has been suggested earlier by Berry and has been described in [10], is to unroll the strongly connected regions of the Conditional Control Flow Graph; Esterel’s constructive semantics guarantees that all unknown inputs to these strongly connected regions can be set to arbitrary, known values without changing the meaning of the program.

As it turns out, the transformation we are proposing here also makes use of this property of constructiveness to resolve cycles; however, unlike the approaches suggested earlier [11, 12], it does so at the source code level. Hence this makes it possible to compile originally cyclic programs using for example the existing efficient compilers that implement event-driven simulators. Furthermore, the experimental results indicate that this transformation can also improve the code resulting from the techniques that can already handle cyclic programs, such as the net-list approach employed by the V5 compiler. It also turns out that the compilation itself can be sped up by transforming cyclic programs into acyclic ones first.

The compilation of Esterel can not only be complicated by cyclic dependencies, but also by *signal reincarnation*, also known as *schizophrenia* [2]. An efficient cure for schizophrenia in Esterel has been proposed by Tardieu [21]. Since it is also based on source code transformation and orthogonal to our approach to handle cycles, it is suited to be sequentially applied to Esterel programs together with the transformation proposed here.

## 2 The Basic Transformation Algorithm

Figure 5 introduces the notation we will use for our transformation. Figure 6 presents the algorithm for transforming cyclic Esterel programs into acyclic programs. The algorithm is applicable to programs with cycles that involve pure signals only. The following section will discuss each transformation step along with its worst-case increase in code size.

## Basics

$\mathbf{N}$ : Set of natural numbers

For  $n \in \mathbf{N} : \mathbf{N}_n =_{def} \{i \in \mathbf{N} \mid i \leq n\}$

$P$ : Given Esterel Program

$\Sigma$ : Set of signals used in  $P$

## Cycle Terminology

$\mathcal{E}_i = \langle \sigma_i, \mathcal{G}_i \rangle$ , with  $i \in \mathbf{N}$ : Guarded emit

$\mathcal{G}_i$ : Boolean expression (“Guard”) involving signals  $\Sigma_i \subseteq \Sigma$

$\sigma_i \in \Sigma$ : Signal emitted in guarded emit

$\mathcal{C} = \{\mathcal{E}_i \mid i \in \mathbf{N}_l\}$ , with  $l \in \mathbf{N}$ : Cycle of length  $l$

$\sigma_{(i \bmod l)+1} \in \Sigma_i$ : Cycle property

## Signals

$\Sigma_{\mathcal{C}}$ : Set of original cycle signals

$\sigma'_i$ : Fresh signal used to replace emission of  $\sigma$  in  $\mathcal{E}_i$

$\Sigma'_{\mathcal{C}}$ : Set of fresh cycle signals

$ST_i$ : Fresh state signal (used to indicate testing of guarded emit)

$ST = \{ST_i \mid i \in \mathbf{N} \cup \{0\}\}$ : Set of state signals

Figure 5: Notation.

**Input:** Program  $P$ , potentially containing cycles

**Output:** Modified program  $P''$ , without cycles

1. Check constructiveness of  $P$ . If  $P$  is not constructive: **Error**.
2. Preprocessing of  $P$ :
  - (a) If  $P$  is composed of several modules, instantiate them into one flat **main** module.
  - (b) Expand derived statements that build on the kernel statements.
  - (c) Rename locally defined signals to make them unique and lift the definitions up to the top level. Furthermore, eliminate signal reincarnation.
  - (d) Transform **suspend** into equivalent **present**/trap statements.
  - (e) Rename trap names making them unique.
3. Introduce state signals:
  - (a) Add boot register as a new global signal **ST\_0** and add “emit **ST\_0**;” to the start of the program body.
  - (b) Enumerate all **pause** and **||** statements starting from 1 and do for all  $\text{pause}_i$  and  $\text{||}_i$ :
    - Globally declare a new signal **ST\_** $i$ .
    - Replace  $\text{pause}_i$  by “pause; emit **ST\_** $i$ .”
    - Replace “p **||** $_i$  q” by “[p **||** q]; emit **ST\_** $i$ .”
4. If  $P$  does not contain cycles: **Done**.  
Otherwise: Select a cycle  $\mathcal{C}$ , of length  $l$ .
5. Transform  $P$  into  $P'$ ; do for all  $\sigma_i \in \mathcal{C}$ :
  - (a) If  $\sigma_i$  is an output signal in the module interface, then add  $\sigma'_i$  to the list of output signals; otherwise, globally declare a new signal  $\sigma'_i$ .
  - (b) Replace “emit  $\sigma_i$ ” by “emit  $\sigma'_i$ .”
  - (c) Replace tests for  $\sigma_i$  by tests for “( $\sigma_i$  or  $\sigma'_i$ ).”
6. Transform (still cyclic)  $P'$  into (acyclic)  $P''$ :
  - (a) For all  $\sigma'_i \in \Sigma'_\mathcal{C}$  determine replacement expressions  $\mathcal{R}_i$ .
  - (b) Select some cycle signal  $\sigma'_i \in \Sigma'_\mathcal{C}$ .
  - (c) Iteratively transform  $\mathcal{R}_i$  to  $\mathcal{R}_i^*$  by replacement of all signals  $\sigma'_j \in (\Sigma'_\mathcal{C} \setminus \sigma'_i)$  by their expressions  $\mathcal{R}_j$ .
  - (d) Replace  $\sigma'_i$  in  $\mathcal{R}_i^*$  by **false** (or **true**) and minimize result.  
Now  $\mathcal{R}_i^*$  does not involve any cyclic signals.
  - (e) Replace all tests for  $\sigma'_i$  in  $P'$  by  $\mathcal{R}_i^*$ .
7. Goto Step (4), treat  $P''$  now as  $P$ .

Figure 6: Transformation algorithm, for pure signals.

**Step (1):** The constructiveness of the Esterel program is a precondition for our transformation. This analysis can be performed using the methods developed by Shiple et al. and by Berry [20, 2]; one available implementation is offered by the v5 compiler [13].

Conversely, the fact that our transformation is valid if and only if the transformed program is constructive can be exploited to employ this transformation to aid in constructiveness analysis in the first place; see also the comments on Step (6d) on page 12.

**Step (2):** The core algorithm is only applicable to Esterel programs restricted in certain ways, requiring the following preprocessing steps:

**Step (2a):** The expansion of modules is a straightforward textual replacement of module calls by their respective body. No dynamic runtime structures are needed, since Esterel does not allow recursions.

The complexity of this module expansion can reach exponential growth of code size, but this expansion is done by every Esterel compiler and not a special requirement of this transformation algorithm.

**Step (2b):** Regarding the statements handling signals, the transformation algorithm is expressed in terms of Esterel kernel statements. Therefore statements that are derived from `emit`, `present`, or `suspend` must be reduced to these statements.

One derived statement is replaced by a fixed construct of kernel statements, therefore the complexity of this step is a constant factor on the number of statements in the program.

**Step (2c):** We have to eliminate locally defined signals because replacement expressions for signals computed by our algorithm could carry references to local signals out of their scope. (Note that the programmer may still freely use local signal declarations.) Furthermore, our method of finding replacement expressions assumes that signals are unique, i. e., not re-incarnated. A simple approach to eliminate reincarnation is based on loop-unrolling, which results in a potentially exponential increase in code size; using other techniques, this can be reduced to a quadratic increase [2], or even lower complexity by the introduction of a `gotopause` statement [21].

**Step (2d):** Program fragments of the form

```
suspend  
  p  
when S
```

where `p` denotes the suspended body and `S` the suspension condition, are replaced by just the body `p`, where all `pause` statements inside `p` are replaced by “`await not S.`” This transformation emulates the behavior of `suspend` by explicitly checking the suspension condition at the start of each instant. However, as the `await` statement is a derived statement, we have to transform it further into kernel statements; “`await not S`” then becomes:

```

trap T in
  loop
    pause;
    present S else exit T end
  end
end

```

An example program with `suspend` statements is discussed in Section 6.1 on page 21. The complexity of this transformation is proportional to the number of `pause` statements inside `suspend` statements.

**Step (2e):** Now there may be multiple instances of the same trap name `T`. This constitutes a valid Esterel program; however, it simplifies the subsequent transformation to have unique trap names.

**Steps (3):** The introduction of state signals makes the current state of the program available to signal expressions. (Note that many of the signals may be eliminated again by subsequent optimizations, see Section 5.) Each `pause` statement is supplemented with the emission of a unique signal `STi`. The first state signal `ST0` is emitted at program start. `ST0` resembles the boot register in the circuit representation of Esterel programs. Additionally all parallel operators (`||`) are extended by the emission of a state signal at termination.

The number of additional state signals and signal emissions is proportional to the number of `pause` statements and parallel operators in the program and therefore proportional to the size of the program.

**Step (4):** Cycles in the program are identified by building a graph representing the control flow dependencies between `present` tests and signal emissions. That directed graph can be used to search for cyclic dependencies in the Esterel program. Only the signals that are part of the cycle are of further interest.

If there is more than one cycle present in the program, then Steps (5) through (6) are performed for each cycle.

**Steps (5a/5b):** This step splits each cycle signal  $\sigma_i$  into two signals  $\sigma_i$  and  $\sigma'_i$ . The signal with the original name  $\sigma_i$  is emitted outside the cycle, a signal with a new name  $\sigma'_i$  is emitted as part of the cycle. The motivation of this step is to distinguish between emissions from inside and outside the cycle; the aim of the replacement expression (see Step (6a)) is to replace emissions inside the cycle. In a way, this introduction of fresh signals, which are emitted exclusively in the cycle, is akin to Static Single Assignment (SSA) [9].

For each signal in the program, at most one replacement signal is added, thus the complexity of this step is a constant factor of the program size.

**Step (5c):** All tests for cycle signals in the original program are extended by tests for their replacement signals. Using the SSA analogy, this corresponds to a  $\phi$ -node [9].

Each changed signal test is expanded by an expression of constant size, therefore we get a constant factor on the number of signal test expressions in the program.

**Step (6a):** The computation of replacement expressions is described in detail in Sec-

tion 3 on page 13.

**Step (6b):** One signal in the set of cyclic signals must be selected as a point to break the cyclic dependency. Any signal in the cycle will work; the actual selection can be based on the smallest replacement expression computed in the next step.

**Step (6c):**  $\mathcal{R}_i$  contains references to other cycle signals  $\sigma'_j$ . These are recursively replaced by their respective expressions  $\mathcal{R}_j$  into  $\mathcal{R}_i^*$ . This unfolding of expressions is performed until only  $\sigma'_i$  and non-cyclic signals are referenced in  $\mathcal{R}_i^*$ .

The complexity of the replacement expressions depends on the length of the cycle, because the length of the cycle dictates the number of replacement iterations needed to eliminate all but the first cycle signals in the guard expression. The length of the cycle and the size of each replacement are limited by the number of signals in the program. So there is a quadratic dependency of the size of the replacement expression to the program size. The number of times the replacement expression will be inserted in the program is likewise dependent on the program size. Thus the growth in program size for one cycle is of cubic complexity.

**Step (6d):** Since the program is known to be constructive, it follows that  $\sigma'_i$  in  $\mathcal{R}_i^*$  must not have any influence on the evaluation of  $\mathcal{R}_i^*$ . Therefore we can replace  $\sigma'_i$  in  $\mathcal{R}_i^*$  by any constant value (**true** or **false**). Now  $\mathcal{R}_i^*$  contains only non cyclic signals. It is important to replace just  $\sigma'_i$  and not  $\sigma_i$ , because  $\sigma_i$  is emitted outside the cycle and therefore not part of the cycle.

The remaining **true** or **false** values must be used to minimize the expression, because some Esterel compilers do not support those boolean constants.

$\sigma'_i$  has no influence on  $\mathcal{R}_i^*$  for all reachable signal states and control states if the program is constructive. This does not necessarily hold for *all* states of signals in  $\mathcal{R}_i^*$ , but only for those reached at runtime at the evaluation of  $\mathcal{R}_i^*$ . This follows from the iterative process of signal replacement in Step (6c) which is equivalent to a symbolic version of a three-valued fix point iteration proposed by Malik [17] and Shiple *et al.* [20]. Note that if the result of the derived expression  $\mathcal{R}_i^*$  is independent of  $\sigma'_i$  for all *reachable* signal combinations at the control point where  $\mathcal{R}_i^*$  is evaluated, then the program is constructive with regard to signal  $\sigma_i$  at that control point. As mentioned above, this observation could be used to aid constructiveness analysis, as this would eliminate the need to perform a fixed-point iteration; nevertheless, to determine the reachable control flow and signal space remains a nontrivial problem.

**Step (6e):** The last transformation step in the algorithm replaces every occurrence of  $\sigma'_i$  in **present** tests by it's replacement expression  $\mathcal{R}_i^*$ . Now we have replaced one signal of the cycle by an expression which is not part of the cycle. Therefore we have *broken* the current cycle  $\mathcal{C}$ .

**Step (7):** The transformation algorithm must be repeated for each cycle, and the upper limit of cycles to resolve is the number of signals in the program.

Overall, a very conservative estimate results in a code size of  $\mathcal{O}(n^4)$ , where  $n$  is the source program size after module expansion and elimination of signal reincarnations; however, we expect the typical code size increase to be much lower. In fact, we often experience an actual reduction in source size, as the transformation often offers opti-

|  |   |
|--|---|
| <pre> <b>present</b> l1 <b>then</b>   <b>present</b> l2 <b>else</b>     <b>emit</b> A   <b>end</b> <b>end</b> <b>end</b>    <b>present</b> A <b>then</b>   <b>emit</b> B <b>end</b> </pre> | <pre> <b>present</b> l1 <b>then</b>   <b>present</b> l2 <b>else</b>     <b>emit</b> A   <b>end</b> <b>end</b> <b>end</b>    <b>present</b> [l1 <b>and not</b> l2] <b>then</b>   <b>emit</b> B <b>end</b> </pre> |
| (a)  | (b)   |

Figure 7: Replacing the signal test for A by their emission context.

mization opportunities where statements are removed. As for the size of the generated object code, here the experimental results (Section 7) also demonstrate that typically the transformation results in a code size reduction.

### 3 Computing the replacement expressions

One step towards breaking cyclic dependencies in Esterel programs is to replace within the conditions of **present** tests the name of a certain signal by an expression (Step (6a) of the algorithm). That expression is derived from the control flow contexts of the program where the signal is set by **emit** statements.

The Logical Behavioral Semantics rules [2] serve as a base to derive the control flow context for a given Esterel program. Our rules are direct derivations from those rules with the aim of an easy implementation.

The main objective of the rules is to get replacement expressions for all signals. The replacement expression describes the signal context of each emission for that signal. Therefore as a prerequisite the signal context of each **emit** statement is needed.

The rules to implement both tasks operate on an Esterel Program  $P$  and two sets:

- $C$  : Current signal context expression;
- $R$  : Accumulation of replacement expressions for signals.

The rules are implemented in two functions:

- $\mathcal{R}: P \times C \rightarrow R$   
This function returns a mapping of signal names to their signal contexts at the point of their emission.
- $\mathcal{C}: P \times C \rightarrow C$   
 $\mathcal{C}$  takes the signal context delivered by previous statements, computes the signal context for sub statements, and returns the signal context for following statements.

$$\begin{array}{lll} \text{emit } S & \begin{array}{l} \mathcal{R}(!S, C) \\ \mathcal{C}(!S, C) \end{array} & \begin{array}{l} = \{(S, C)\} \\ = C \end{array} \end{array} \quad (1)$$


---

$$\begin{array}{lll} \text{present } S \text{ then} & & \\ \quad p & \mathcal{R}(S?p, q, C) & = \mathcal{R}(p, C \wedge S) \cup \mathcal{R}(q, C \wedge \bar{S}) \\ \text{else} & \mathcal{C}(S?p, q, C) & = \mathcal{C}(p, C \wedge S) \vee \mathcal{C}(q, C \wedge \bar{S}) \\ \quad q & & \\ \text{end} & & \end{array} \quad (2)$$


---

$$\begin{array}{lll} \text{nothing} & \begin{array}{l} \mathcal{R}(0, C) \\ \mathcal{C}(0, C) \end{array} & \begin{array}{l} = \emptyset \\ = C \end{array} \end{array} \quad (3)$$


---

$$\begin{array}{lll} \text{pause;} & \mathcal{R}(1;!ST_i, C) & = \emptyset \\ \text{emit } ST_i: & \mathcal{C}(1;!ST_i, C) & = ST_i \end{array} \quad (4)$$


---

$$\begin{array}{lll} \text{exit } T & \begin{array}{l} \mathcal{R}(k, C) \\ \mathcal{C}(k, C) \end{array} & \begin{array}{l} = \{(T, C)\} \\ = \text{false} \end{array} \end{array} \quad (5)$$


---

$$\begin{array}{lll} \text{trap } T \text{ in} & \mathcal{R}(\{p\}, C) & = \mathcal{R}(p, C) \\ \quad p & \mathcal{C}(\{p\}, C) & = \mathcal{C}(p, C) \vee \bigvee_{(T, c_i) \in \mathcal{R}(p, C)} c_i \\ \text{end} & & \end{array} \quad (6)$$


---

$$\begin{array}{lll} p; q & \begin{array}{l} \mathcal{R}(p; q, C) \\ \mathcal{C}(p; q, C) \end{array} & \begin{array}{l} = \mathcal{R}(p, C) \cup \mathcal{R}(q, \mathcal{C}(p, C)) \\ = \mathcal{C}(q, \mathcal{C}(p, C)) \end{array} \end{array} \quad (7)$$


---

$$\begin{array}{lll} \text{loop} & \mathcal{R}(p^*, C) & = \mathcal{R}(p, C \vee \mathcal{C}(p, C)) \\ \quad p & \mathcal{C}(p^*, C) & = \text{false} \\ \text{end} & & \end{array} \quad (8)$$


---

$$\begin{array}{lll} [p \parallel q]; & \mathcal{R}((plq); !ST_i, C) & = \mathcal{R}(p, C) \cup \mathcal{R}(q, C) \\ \text{emit } ST_i & \mathcal{C}((plq); !ST_i, C) & = ST_i \end{array} \quad (9)$$


---

$$\begin{array}{lll} \text{signal } S \text{ in} & \mathcal{R}(p \setminus S, C) & = \mathcal{R}(p, C) \\ \quad p & \mathcal{C}(p \setminus S, C) & = \mathcal{C}(p, C) \\ \text{end} & & \end{array} \quad (10)$$

Figure 8: Equations to determine replacement expressions for signals



$$\begin{array}{c}
\frac{s^+ \in E \quad p \xrightarrow[E]{E',k} p'}{\quad} \\
s^?p,q \xrightarrow[E]{E',k} p' \\
\text{(a) present+}
\end{array}
\qquad
\begin{array}{c}
\frac{s^- \in E \quad q \xrightarrow[E]{E',k} q'}{\quad} \\
s^?p,q \xrightarrow[E]{E',k} q' \\
\text{(b) present-}
\end{array}$$

Figure 9: Logical Behavioral Semantics of the **present** statement

These functions are computed by structural induction over their first argument (an Esterel program); the corresponding definitions for each kernel statement are given in Figure 8. To determine the replacement expressions for all signals in a program  $P$ , we compute  $R := \mathcal{R}(P, \text{ST}_0)$ , where  $\text{ST}_0$  denotes the boot signal, present only at startup in the very first instant. The result of  $\mathcal{R}$  will be a set of pairs. Each pair consists of a signal name and a signal expression (condition). The expressions describe in which signal context each signal is emitted. Now the expressions for the same signals can be conjuncted to yield a single replacement expression for the emission of each signal.

As an example to illustrate how the definitions of  $\mathcal{R}$  and  $\mathcal{C}$  correspond to the behavioral semantics, consider the **present** statement. The two SOS rules from the Logical Behavioral Semantics for the **present** statement, given in Figure 9 [2], select the rule to apply based on the presence of the condition signal  $s$  and the resulting control flow. The selected rule will add signal emissions etc. to the resulting context. The corresponding equations for  $\mathcal{R}$  and  $\mathcal{C}$  (2) consider both possible control flow paths, and both paths may add signal emissions to  $R$ ; however, each signal emission is tied to the condition for that part, thus reflecting the original semantics.

Trap signals are treated like regular signals. Rule (5) adds the current signal context as an emission context for the trap signal to  $\mathcal{R}$ . Rule (6) takes all signal contexts for the trap signal from  $\mathcal{R}$  and adds them to the signal context after leaving the trap statement block.

## 4 Example Transformations

We will illustrate the transformation algorithm described in Section 2 by applying it to the `PAUSE_CYC` and `TR3_CYC` examples.

### 4.1 Transforming `PAUSE_CYC`

The algorithm is applied to the example `PAUSE_CYC` in Figure 2(a) on page 4, which is transformed into the acyclic program `PAUSE_ACYC` in Figure 2(c). The transformation of the program `DRIVER_CYC` in Figure 3(a), page 5, into `DRIVER_ACYC` in Figure 3(b) is similar.

Step (1): `PAUSE_CYC` is cyclic but nevertheless constructive, because a **pause** statement separates the execution of both parts of the cycle.

Steps (2a) to (2e) do not apply to PAUSE\_CYC.

Step (4): PAUSE\_CYC contains one cycle.  $\mathcal{C} = \{\langle A, B \rangle, \langle B, A \rangle\}$ .

Steps (3) and (5): To prepare the removal of the cycle, we first transform PAUSE\_CYC into the equivalent program PAUSE\_PREP, shown in Figure 2(b). It differs from PAUSE\_CYC in the introduction of state signals ST\_0 to ST\_2 and in the fact that the signals carrying the cycle (A and B) have been replaced by fresh signals A\_ and B\_, which are only emitted within the cycle. All tests for A and B in the original program are replaced by tests for [A or A\_] and [B or B\_], respectively.

Step (6a): The computation of replacement expressions for A\_ and B\_ according to Section 3 results in:

$$A_ = ST_1 \wedge (B \vee B_) \quad (11)$$

$$B_ = ST_0 \wedge (A \vee A_) \quad (12)$$

The equations for each signal now refer to other cycle signals; note that we consider A and B not cycle signals anymore, as they are not emitted within the cycle anymore. The similarity to a system of linear equations is apparent and we solve the equations likewise:

Step (6b): In PAUSE\_PREP, we arbitrarily select A\_ as the signal to break the cycle.

Step (6c): To replace B\_ in Equation (11), substituting (12) into (11) results in:

$$A_ = ST_1 \wedge (B \vee (ST_0 \wedge (A \vee A_))). \quad (13)$$

This is now an equation which expresses the cycle signal A\_ as a function of itself and other signals that are not part of the cycle; so we have unrolled the cycle.

Step (6d): We could now simulate (13) using three-valued logic; however, here we make use of the constructiveness of the program, which guarantees monotonicity. This means that a more defined input always produces an equal or more defined output. Hence, if the program is known to never produce undefined outputs, we can set all unknown inputs (such as A\_ in this case) to arbitrary, known values without changing the meaning of the program [10]. Applying this to Equation (13) yields for  $A_ = \text{false}$  (absent):

$$A_ = ST_1 \wedge (B \vee (ST_0 \wedge A)). \quad (14)$$

Similarly, for  $A_ = \text{true}$  (present):

$$A_ = ST_1 \wedge (B \vee ST_0). \quad (15)$$

We now have derived two equally valid replacement expressions for A\_, which do not involve any cycle signal.

Step (6e): Finally we are ready to break the cycle in PAUSE\_PREP. For that, we have to replace the signal selected in Step (6b)—in the cycle—by an expression that does not use any of the cycle signals, without changing the meaning of the program.

Substituting (15), the simpler of these expressions, for A\_ in PAUSE\_PREP yields the now acyclic program PAUSE\_ACYC shown in Figure 2(c).

```

module TR3_ACYC:

input R1, R2, R3;
output G1, G2, G3;

signal ST_0, ST_1, ST_2, ST_3,
        ST_4, ST_5, ST_6, ST_7,
        ST_8, ST_9, ST_10 in
    emit ST_0;
signal P2, P3,
        % P1 deleted
        T1, T2, T3
in
[
    emit T1
    ||
    loop % STATION1
    present
    [T1 or (ST_0 or ST_7)
    and (T3 or (ST_0 or ST_4)
    and (T2 or (ST_0 or ST_1)
    and not R1)
    and not R2)
    and not R3] then
        present R1 then
            emit G1
        else
            emit P2
        end
    end;
    pause; emit ST_1;
end loop
    ||
    loop
    present T1 then
        pause; emit ST_2;
        emit T2
    else
        pause; emit ST_3;
    end
end
    ||
    loop % STATION2
    present [T2 or P2]
    then
        present R2 then
            emit G2
        else
            emit P3
        end
    end;
    pause; emit ST_4
end loop
    ||
    loop
    present T2 then
        pause; emit ST_5;
        emit T3
    else
        pause; emit ST_6
    end
end
    ||
    loop % STATION3
    present [T3 or P3]
    then
        present R3 then
            emit G3
        % else branch
        % deleted
    end
    end;
    pause; emit ST_7
end loop
    ||
    loop
    present T3 then
        pause; emit ST_8;
        emit T1
    else
        pause; emit ST_9
    end
    end
]; emit ST_10
end module

```

Figure 10: Non cyclic Token Ring Arbiter.

## 4.2 Transforming the Token Ring Arbiter

Before transforming program TR3\_CYC from Figure 4 into the acyclic program TR3\_ACYC shown in Figure 10, we can apply an optimization. There are no tests for the cycle signals outside of the cycle, so we do not need fresh cycle signals either. This and other optimizations are explained further in Section 5.

We now select signal P1 to break the cycle. We can compute the expression to replace P1 in the test in STATION1 as follows:

$$P2 = (ST\_0 \vee ST\_1) \wedge (T1 \vee P1) \wedge \overline{R1}, \quad (16)$$

$$\begin{aligned} P3 &= (ST\_0 \vee ST\_4) \wedge (T2 \vee P2) \wedge \overline{R2} \\ &= (ST\_0 \vee ST\_4) \wedge (T2 \vee (ST\_0 \vee ST\_1) \wedge (T1 \vee P1) \wedge \overline{R1}) \wedge \overline{R2}, \end{aligned} \quad (17)$$

$$\begin{aligned} P1 &= (ST\_0 \vee ST\_7) \wedge (T3 \vee P3) \wedge \overline{R3} \\ &= (ST\_0 \vee ST\_7) \wedge (T3 \vee (ST\_0 \vee ST\_4) \wedge (T2 \vee (ST\_0 \vee ST\_1) \\ &\quad \wedge (T1 \vee P1) \wedge \overline{R1}) \wedge \overline{R2}) \wedge \overline{R3}. \end{aligned} \quad (18)$$

Equation (18) now again expresses a cycle carrying signal (P1) as a function of itself and other signals that are outside of the cycle. Again we can employ the constructiveness of TR3\_CYC to replace P1 in this replacement expression by either true or false. Setting P1 to false yields:

$$P1 = (ST\_0 \vee ST\_7) \wedge (T3 \vee (ST\_0 \vee ST\_4) \wedge (T2 \vee (ST\_0 \vee ST\_1) \wedge T1 \wedge \overline{R1}) \wedge \overline{R2}) \wedge \overline{R3}. \quad (19)$$

Setting P1 to true yields:

$$P1 = (ST\_0 \vee ST\_7) \wedge (T3 \vee (ST\_0 \vee ST\_4) \wedge (T2 \vee (ST\_0 \vee ST\_1) \wedge \overline{R1}) \wedge \overline{R2}) \wedge \overline{R3}. \quad (20)$$

The shorter expression (20) is applied when transforming TR3\_CYC. The other transformation steps are fairly straightforward.

The replacement expression is fairly complex, but close inspection yields an optimization: The expression  $(ST\_0 \vee ST\_7)$  is contained in (20): The state signal ST\_0 is emitted in the first instant and ST\_7 is emitted in all instants but the first one. In a disjunction they will always return true. Therefore the expression can be replaced statically by true. The same holds for  $(ST\_0 \vee ST\_4)$  and  $(ST\_0 \vee ST\_1)$ .

With this optimization (20) can be reduced to:

$$P1 = (T3 \vee (T2 \vee \overline{R1}) \wedge \overline{R2}) \wedge \overline{R3}. \quad (21)$$

Further optimization opportunities are discussed in the following Section 5.

## 5 Optimizations

Figure 6 and Figure 8 contain the algorithm and equations for replacement expressions in its basic form without any additional optimizations. In the following we will present some possible improvements to the algorithm, but only the first one is included in the current implementation.

## 5.1 Replacement expression for present

The most important optimization refines the treatment of the **present** statement in Equation (2). Consider the following program fragment:

```

pause; emit ST_3;
present S then
  emit A
else
  emit B
end;
emit C

```

The application of the rules listed in Figure 8 on page 14 would result in a replacement expression for signal  $C = (ST\_3 \wedge S) \vee (ST\_3 \wedge \bar{S})$ . It is obvious that this can be simplified to  $C = ST\_3$ . Or more generally:

Equation (2):

$$\mathcal{C}((S?p,q), C) = \mathcal{C}(p, C \wedge S) \vee \mathcal{C}(q, C \wedge \bar{S})$$

can be simplified to

$$\mathcal{C}((S?p,q), C) = C$$

if

$$\mathcal{C}(p, C \wedge S) = C \wedge S \quad \text{and} \quad \mathcal{C}(q, C \wedge \bar{S}) = C \wedge \bar{S}$$

holds.

This optimization alone yields a considerable reduction in the size of replacement expressions in the current implementation.

## 5.2 Replacing state signal tests by constants

Another optimization (currently not implemented) would be to try to determine which state signals are always present or absent in a replacement expression. For example, the program PAUSE\_ACYC can be optimized into the program PAUSE\_OPT shown in Figure 2(d) on page 4 by taking the reachable presence status of the signals ST\_0 and ST\_1 into account.

The replacement expression  $\mathcal{R}_i^*$  (Step (6e) of the algorithm) may reference some state signal  $ST_j \in STs$  that can be shown to be always present or absent:

1. If  $\mathcal{R}_i^*$  replaces  $\sigma'_i$  in  $\mathcal{R}_k$ , and we know that at this location in the program,  $ST_j$  must always be present, then we can replace  $ST_j$  by the constant **true** in  $\mathcal{R}_i^*$ .

In the program PAUSE\_ACYC, this applies to the state signal ST\_0 in the replacement expression (ST\_1 and (B or ST\_0)), which we therefore can simplify to (ST\_1 and B).

More generally, we could replace a state signal by the constant **true** if we knew that it must be emitted in every instant. As it turns out, there cannot be any state signals that fulfill this condition by themselves; the boot state signal ST\_0 is only present in the initial instant, and all other state signals are emitted only after a **pause** statement, and hence cannot be emitted in the initial instant. However, another optimization is possible:

2. If a state signal is emitted in every instant except for the initial instant, we can replace it with `pre(tick)`. This replacement eliminates the need to introduce the state signal into the program.

In the Arbiter, for example, the state signals `ST_1`, `ST_4`, and `ST_7` are the only state signals emitted in loops that run concurrently to the rest of the program—hence, as loops must not be instantaneous, they must be emitted at every iteration and are present at every instant except the initial one.

3. Tests for “`ST_0` or `pre(tick)`” can be replaced by `true`.

This applies for example to the Token Ring Arbiter, where we know that all guarded emits that constitute the cycle are evaluated in every instant of the program. Hence this rule, together with the previous rule, leads to the simplified replacement expression already stated in Equation (21).

4. Correspondingly, it may also be the case that a state signal is always absent when tested in some replacement expression  $\mathcal{R}_i^*$ . In particular, this is the case when we have a false cycle.

In the program `PAUSE_ACYC`, this applies to the state signal `ST_1`; due to the `pause` statement between the evaluation of the replacement expression and the emission of `ST_1`, we can set `ST_1` to `false` in the replacement expression. In this case, this reduces the whole replacement expression to `false`; therefore, the “[`A` or (`ST_1` and (`B` or `ST_0`))]” from `PAUSE_ACYC` gets reduced to just `A` in `PAUSE_OPT`.

### 5.3 Eliminating emission of state signals

If all tests for a state signal are replaced by constants, the state signal is no longer needed and therefore does not need to be emitted any more.

In the program `PAUSE_ACYC`, this applies to both `ST_0` and `ST_1`, we can therefore drop the corresponding emit in the optimized `PAUSE_OPT`.

### 5.4 Absence of External Emissions of Cycle Signals

If a cycle signal  $\sigma_i$  is not emitted outside of the cycle, we do not need to generate a fresh signal  $\sigma'_i$ , but can instead just use  $\sigma_i$ . In this case, one may skip Step (5).

This is the case in the Arbiter, where the signals carrying the cycle (`P1/P2/P3`) are not emitted outside of the cycle.

### 5.5 Absence of External Tests of Cycle Breaking Signal

If the signal  $\sigma'_i$  that is selected in Step (6b) to break the cycle is not tested outside of the cycle, this means that after replacing the tests for  $\sigma'_i$  within the cycle (Step 6e) by

$\mathcal{R}_i^*$ , the signal  $\sigma'_i$  is not tested anywhere in the program. One can therefore eliminate its emission.

This also applies to the Arbiter, where signal P1, which we replaced within the cycle, becomes superfluous. We can therefore eliminate the "emit P1," and the enclosing else branch.

## 5.6 Simplification of External Tests

Depending on how often one must replace a particular signal  $\sigma_i$  in Step (5c) by the expression " $(\sigma_i \text{ or } \sigma'_i)$ ," it may be beneficial to introduce another fresh signal  $\sigma''_i$ . This signal must be emitted whenever  $\sigma_i$  or  $\sigma'_i$  are present, for example using a new globally parallel statement of the form "every [ $\sigma_i$  or  $\sigma'_i$ ] do emit  $\sigma''_i$  end." Then it suffices to replace tests for  $\sigma_i$  by tests for  $\sigma''_i$ .

# 6 Further Example Transformations

Figure 11 gives another program with a cycle involving signals A and B. However, the cycle is only present at certain instants; the guarded emit of B takes place every 3rd instant, whereas the guarded emit of A takes place every 5th instant, hence the cycle is active only every 15th instants. Here the transformation into an acyclic version requires the emission of the signal ST\_2, which indicates the evaluation of the guard dependency with sink B. As an optimization, due to the invariant "T1 or T2 = true," which is ensured by the third parallel thread, it suffices to just replace the guard in the guard dependency involving A in the transformed program by "A or ST\_2".

## 6.1 Suspend

So far we presented only cycles with a **present** test as a guard for an **emit** statement. Another way to influence the execution of **emit** is the **suspend** statement. A complication with **suspend** is that, unlike with **present**, one cannot easily generate a signal that is emitted unconditionally whenever the guard of a **suspend** is evaluated. The transformation algorithm therefore first transforms the **suspend** statements into equivalent **present/trap** statements, in Step (2d).

As an example, consider the program SUSP\_CYC in Figure 12(a). The program again contains a cyclic dependency on the signals A and B, the emission of each signal is inhibited by the presence of the other signal.

Applying Steps (2d), (3b) results in the preprocessed program SUSP\_PREP shown in Figure 12(b). The end result, after applying the whole transformation algorithm, is SUSP\_ACYC in Figure 12(c).

```

module PAUSES_CYC:
input A, B;
output C, D;

signal T1, T2 in
  loop
    pause; pause;
    pause;
    present [A or T1]
    then
      emit B;
      emit C
    end
  end
  ||
  loop
    pause; pause;
    pause; pause;
    pause;
    present [B or T2]
    then
      emit A;
      emit D
    end
  end
  ||
  loop
    emit T1;
    pause;
    emit T2;
    pause
  end
end
end module

```

(a)

```

module PAUSES_PREP:
input A, B;
output C, D;

signal ST_1, ST_2 in
  signal T1, T2 in
    loop
      pause; pause;
      pause; emit ST_1;
      present [A or A_ or T1]
      then
        emit B_;
        emit C
      end
    end
  ||
  loop
    pause; pause;
    pause; pause;
    pause; emit ST_2;
    present [B or B_ or T2]
    then
      emit A_;
      emit D
    end
  end
  ||
  loop
    emit T1;
    pause;
    emit T2;
    pause
  end
end
end module

```

(b)

```

module PAUSES_ACYC:
input A, B;
output C, D;

signal ST_1, ST_2 in
signal T1, T2 in
  loop
    pause; pause;
    pause; emit ST_1;
    present [A or
  ST_2 and (B or ST_1 or T2)
    or T1] then
      then
        emit B_;
        emit C
      end
    end
  ||
  loop
    pause; pause;
    pause; pause;
    pause; emit ST_2;
    present [B or B_ or T2]
    then
      emit A_;
      emit D
    end
  end
  ||
  loop
    emit T1;
    pause;
    emit T2;
    pause
  end
end
end module

```

(c)

Figure 11: Example requiring state signals which can not be eliminated: (a) Cyclic dependency between signals A and B, (b) introduction of state signals and signal separation (unused state signals are left out for brevity), (c) acyclic program obtained by replacing A\_ with an expression.



```

module SUSP_CYC:
output A,B;

  pause;
  suspend
    pause;
    emit A
  when B
  ||
  suspend
    pause;
    emit B
  when A
end module

```

(a)

```

module SUSP_PREP:
output A,B;

signal ST_0, ST_1,
        ST_2, ST_3, ST_4 in
  emit ST_0;
  [
    pause; emit ST_1;
    trap T1 in
      loop
        pause; emit ST_2;
        present B else
          exit T1
        end
      end;
    end;
    emit A
  ] ||
  trap T2 in
    loop
      pause; emit ST_3;
      present A else
        exit T2
      end
    end;
    emit B
  ]; emit ST_4
end signal
end module

```

(b)

```

module SUSP_ACYC:
output A,B;

signal ST_0, ST_1,
        ST_2, ST_3, ST_4 in
  emit ST_0;
  [
    pause; emit ST_1;
    trap T1 in
      loop
        pause; emit ST_2;
        present B else
          exit T1
        end
      end;
    end;
    emit A
  ] ||
  trap T2 in
    loop
      pause; emit ST_3;
      present
        [ST_2 and not ST_3]
      else
        exit T2
      end
    end;
    emit B
  ]; emit ST_4
end signal
end module

```

(c)

Figure 12: Simple cyclic program with suspend.

```
module VALUE_CYC:
```

```
input S;  
input A : integer , B : integer ;  
output X : integer , Y : integer ;
```

```
present S then  
  present A then  
    emit B(?A)  
  end  
else  
  present B then  
    emit A(?B)  
  end  
end;
```

```
present A then emit X(?A) end;  
present B then emit Y(?B) end  
end module
```

(a)

```
module VALUE_ACYC:
```

```
input S;  
input A : integer , B : integer ;  
output X : integer , Y : integer ;
```

```
signal B_:integer, B__:integer in
```

```
  present S then  
    present A then  
      emit B_(?A)  
    end  
  else  
    present B then  
      emit A(?B)  
    end  
end;
```

```
  present A then emit X(?A) end;  
  present B__ then emit Y(?B__) end
```

```
  ||  
  every B do emit B__(?B) end
```

```
  ||  
  every B_ do emit B__(?B_) end
```

```
end signal
```

```
end module
```

(b)

Figure 13: Esterel program with a cycle on valued signals.

## 6.2 Valued Signals

Figure 13(a) contains an Esterel program `VALUE_CYC` with a (false) cycle on the signals `A` and `B`. Both signals carry values of type `integer`. The pure signal `S` is used to select one of two data flows: from `A` to `B` or vice versa. This results in two guarded emits with reversed signal use. Therefore both guarded emits constitute a cycle. The cycle is *false*, because signal `S` ensures that only one of both emits is active in an instant. After the guarded emits are evaluated, two additional guarded emits copy the values from `A` and `B` to the outputs `X` and `Y`, respectively.

Figure 13(b) contains the program `VALUE_ACYC`, an acyclic transformation of `VALUE_CYC`. Two additional signals are introduced: `B_` and `B_..`. `B_` is the replacement for `B` to break the cycle, and `B_..` is used to represent the state of `B` outside the cycle. Two additional parallel statements forward the values of `B` (from the module interface) and `B_` (from the cycle) to the signal `B_..`.

There are two `emit` statements for the same signal `B_..`. Both are executed in the same instant if `B` is emitted on the module interface and `B_` is emitted in the cycle. Therefore we need a *combine* function if we cannot exclude this case. However, this problem is not introduced by the transformation of the cycle. The original program `VALUE_CYC` needs a combine function, too. Consider `S`, `A` and `B` being present at the module interface. Then the first guarded emit will be executed and `B` will be emitted in the same instant a second time. The same holds for `S` absent, then `A` will be emitted twice. We can compile the program without a combine function if we assert that `A` and `B` are not both present in the same instant. This resolves the problem for `VALUE_ACYC`, too.

## 7 Experimental Results

The proposed algorithm has been implemented in its basic form, so far without support for valued signals, as an extension of the Columbia Esterel Compiler (CEC). For a first experimental evaluation, we have defined several variants of the Token Ring Arbiter:

**TR3:** This is the Token Ring Arbiter with three network stations. The implementation is as in Figure 4.

**TR10:** This is an extension of TR3 from three to ten network stations. The aim is to test the scaling of the algorithm for code size and runtime of the resulting binary.

**TR50, ... TR1000:** These versions contain fifty to one thousand network stations. They are used to estimate the factor of growth for big program sizes and to measure the performance of the transformation itself.

**TR10p:** While the former test cases implemented only the arbiter part of the network without any local activity on the network stations, this test program adds some simple concurrent “payload” activity to each network station to simulate a CPU performing some computations with occasional access to the network bus.

All programs are tested in the originally cyclic and in the transformed acyclic version.

## 7.1 Synthesizing Software

To evaluate the transformation in the realm of generating software, we used six different compilation techniques:

**v5-L:** The publicly available Esterel compiler v5.92 [5, 13]. It is used in this case with option `-L` to produce code based on the circuit representation of Esterel. The code is organized as a list of equations ordered by dependencies. This results in a fairly compact code, but with a comparatively slow execution speed. This compiler is able to handle constructive Esterel programs with cyclic dependencies.

**v5-A:** The same compiler, but with the option `-A`, produces code based on a flat automaton. This code is very fast, but prohibitively big for programs with many weakly synchronized parallel activities. This option is available for cyclic programs, too.

**v7:** The Esterel v7 compiler (available at Esterel Technologies) is used here in version v7\_10i8 to compile acyclic code based on sorted equations, as the v5 compiler.

**v7-O:** The former compiler, but with option `-O`, applies some circuit optimizations to reduce program size and runtime.

**CEC:** The Columbia Esterel Compiler (release 0.3) [7] produces event driven C code, which is generally quite fast and compact. However, this compiler cannot handle cyclic dependencies. Thus it can only be applied to the transformed cyclic programs.

**CEC-g:** The CEC with the option `-g` produces code using computed `goto` targets (an extension to ANSI-C offered by GCC-3.3 [14]) to reduce the runtime even further.

A simple C back-end is provided for each Esterel program to produce input signals and accept output signals to and from the Esterel part. The back-end provides an iteration over 10,000,000 times for the reaction function. These iteration counts result in execution times in the range of about 0.8 to 18 seconds. These times were obtained on a desktop PC (AMD Athlon XP 2400+, 2.0 GHz, 256KB Cache, 1GB Main Memory).

Table 1(a) compares the execution speed and binary sizes of the example programs for the v5, v7, and CEC compilers with their respective options. The v5 compiler is applied both to the original cyclic programs and the transformed acyclic programs. The CEC and v7 compiler can handle only acyclic code.

When comparing the runtime results of the v5 compiler (with sorted equations) for the cyclic and acyclic versions of the token ring arbiter, there is a noticeable reduction in runtime for the transformed acyclic programs. This came as a bit of a surprise. It seems that the v5 compiler is a little bit less efficient in resolving cyclic dependencies in sorted equations. For the automaton code there are only minor differences in runtime.

Table 1 includes the sizes of the compiled binaries, too. All compilers produce code of similar sizes, but with one exception: the v5 compiler produces a very big automaton code for the third token ring example. That program contains several parallel threads which are only loosely related. If someone tries to map such a program on a flat automaton, it is well known that such a structure results in a “state explosion.” Actually, we had to limit the number of parallel tasks in this example to get the program to com-

| Variant                       | Compiler | TR3                 | TR10                | TR10p               |
|-------------------------------|----------|---------------------|---------------------|---------------------|
| cyclic<br>(original)          | v5-L     | 1.55/ 14273         | 5.39/ 21530         | 17.19/ 32244        |
|                               | v5-A     | 0.90/ <b>13041</b>  | 2.58/ <b>16091</b>  | <b>5.26</b> /304095 |
| acyclic<br>(trans-<br>formed) | v5-L     | 1.40/ 14067         | 5.07/ 20188         | 12.16/ 29110        |
|                               | v5-A     | 0.89/ <b>13043</b>  | 2.58/ <b>16093</b>  | <b>5.26</b> /304097 |
|                               | v7       | 1.69/ 14526         | 6.07/ 20255         | 12.34/ 27353        |
|                               | v7-O     | <b>0.53</b> / 13467 | <b>1.87</b> / 16315 | 5.83/ <b>21033</b>  |
|                               | CEC      | 1.80/ 14244         | 6.42/ 22020         | 12.04/ 29579        |
|                               | CEC-g    | 1.09/ 13822         | 3.82/ 20430         | 5.89/ 25461         |

Table 1: Run times (in seconds) and binary sizes (in bytes) of cyclic and acyclic Esterel programs compiled with the v5, v7, and CEC compiler.

pile in reasonable time. While the v5 compiler seems to be competitive with respect to program run times, the binary sizes can reach several times the size of the binaries produced by the other compilers.

For the two token ring arbiter variants without payload, the v7 compiler produces the fastest code. The third token ring example with payload is executed fastest with the v5 compiler in automata mode, but only slightly better than the CEC compiler with computed goto optimization. Nevertheless the huge binary produced by the v5 compiler in automaton mode limits its usefulness.

Table 2 compares the fastest code for our cyclic programs to the fastest code for the transformed acyclic programs. For each test program the relative reduction in runtime is listed.

Table 3 contains the compilation times for the different Esterel compilers to compile the various test programs. The v5 compiler for sorted equations code needs only little time to compile the acyclic versions of the test programs. In fact, it is among the fastest compilers in all three acyclic test cases. When this compiler is applied to cyclic programs, the compilation times are several times slower but within reasonable limits. The transformation times for the acyclic test programs (Table 4) are not included in Table 3, but even if we add the transformation times to the compilation times of the acyclic programs the picture will not change much.

When compiling for automaton code with the v5 compiler, then the compilation time is mostly independent of cyclic and acyclic properties of the compiled program. The compilation times are low for small programs with few states, but drastically higher for programs with many independent, parallel states. The CEC compiler is comparatively slow for small acyclic programs, but the compilation time does not rise that much for more complex programs. The v7 compiler behaves similarly.

As an indication of the cost of the transformation algorithm in terms of processing time and source code increase, Table 3 lists transformation times and program sizes before and after the transformation of the token ring arbiter with 3, 10, 50, 100, 500, and 1000 nodes. The size of the transformed code is nearly proportional with respect to the arbiter network size. The current transformation times show a sub-quadratic effort for the transformation.

|                     | TR3  | TR10 | TR10p |
|---------------------|------|------|-------|
| $\min(T_{cyclic})$  | 0.90 | 2.58 | 5.26  |
| $\min(T_{acyclic})$ | 0.53 | 1.87 | 5.26  |
| <i>reduction</i>    | 41%  | 28%  | 0.0%  |

Table 2: Relative run time reduction from the fastest cyclic version to the fastest version for the acyclic transformation, with  $reduction = 100\% * \left(1 - \frac{\min(T_{acyclic})}{\min(T_{cyclic})}\right)$ .

| Variant                       | Compiler | TR3         | TR10        | TR10p       |
|-------------------------------|----------|-------------|-------------|-------------|
| cyclic<br>(original)          | v5-L     | 0.08        | 0.29        | 1.38        |
|                               | v5-A     | <b>0.01</b> | <b>0.04</b> | 10.86       |
| acyclic<br>(trans-<br>formed) | v5-L     | <b>0.01</b> | 0.06        | <b>0.10</b> |
|                               | v5-A     | <b>0.01</b> | <b>0.04</b> | 10.54       |
|                               | v7       | 0.12        | 0.20        | 0.36        |
|                               | v7-O     | 0.24        | 0.54        | 1.08        |
|                               | CEC      | 0.15        | 0.35        | 0.76        |
|                               | CEC-g    | 0.11        | 0.37        | 0.71        |

Table 3: Compiler run times for Esterel v5, v7, and CEC (in seconds).

| Transformation         | TR3  | TR10 | TR50  | TR100 | TR500  | TR1000 | TR10p |
|------------------------|------|------|-------|-------|--------|--------|-------|
| original size          | 1565 | 3705 | 16348 | 32159 | 162959 | 326470 | 5765  |
| after module expansion | 1370 | 4391 | 22031 | 44092 | 224892 | 450903 | 6995  |
| after cycle transform. | 2108 | 6856 | 34804 | 69920 | 359788 | 723804 | 9736  |
| transform. time        | 0.05 | 0.07 | 0.27  | 0.57  | 5.18   | 17.5   | 0.11  |

Table 4: Transformation times (in seconds) and resulting program sizes (in bytes) for token ring arbiters with 3 to 1000 nodes.

| Variant | Compiler | TR3       | TR10       | TR10p      |
|---------|----------|-----------|------------|------------|
| cyclic  | v5       | 112       | 357        | 759        |
|         | v5       | 108       | 346        | 748        |
|         | v7       | <b>52</b> | <b>171</b> | <b>351</b> |
| acyclic | CEC      | 146       | 468        | 756        |

(a)

| Variant | Compiler | TR3      | TR10      | TR10p     |
|---------|----------|----------|-----------|-----------|
| cyclic  | v5       | 10       | 31        | 55        |
|         | v5       | 10       | 31        | 55        |
|         | v7       | 10       | 31        | 55        |
| acyclic | CEC      | <b>4</b> | <b>11</b> | <b>47</b> |

(b)

| Variant | Compiler | TR3        | TR10       | TR10p      |
|---------|----------|------------|------------|------------|
| cyclic  | v5       | 208        | 745        | 1551       |
|         | v5       | 197        | 645        | 1377       |
|         | v7       | <b>108</b> | <b>360</b> | <b>702</b> |
| acyclic | CEC      | 221        | 725        | 1301       |

(c)

| Variant | Compiler | TR3       | TR10       | TR10p      |
|---------|----------|-----------|------------|------------|
| cyclic  | v5       | <b>82</b> | <b>266</b> | 539        |
|         | v5       | 89        | 299        | <b>524</b> |
|         | v7       | 91        | 315        | 591        |
| acyclic | CEC      | 89        | 313        | 679        |

(d)

Table 5: Comparison of: (a) node count for BLIF output, (b) latch count for BLIF output, (c) sum-of-product (lits(sop)) count for BLIF output, (d) sum-of-product (lits(sop)) - optimized by SIS.

## 7.2 Synthesizing Hardware

To evaluate the effect of our transformation on hardware synthesis, we compared again the results of the v5, v7, and CEC compilers, for the same set of benchmarks as for the software synthesis. Again only v5 can handle the untransformed, cyclic code version; furthermore, v5 is the only compiler that can generate hardware for valued signals. The compilers differ in which hardware description languages they can produce, but a common format supported by all of them is the Berkeley Logic Interchange Format (BLIF), therefore we base our comparisons on this output format.

Table 5(a) compares the number of nodes synthesized. Considering the v5 compiler, there is a noticeable reduction in the number of nodes generated for the Arbiter. When considering the synthesis results of v7 and CEC for the acyclic version of the Arbiter, v7 produces the best overall results, with the node count less than half of v5’s synthesis results for the cyclic variants.

Table 5(b) compares the number of latches needed by the synthesization results. Here the CEC is able to reduce the number of latches considerably.

Table 5(c) compares the number of literals generated. The overall results are similar to the ones for the node count; the transformation has been lowered the literal count for the arbiter.

Table 5(c) compares the number of literals which remain after a SIS [19] optimization.

## 8 Conclusions and future work

We have presented an algorithm for transforming cyclic Esterel programs into acyclic programs. This expands the range of available compilation techniques, and, as to be expected, some of the techniques that are restricted to acyclic programs produce faster

and/or smaller code than is possible with the compilers that can handle cyclic codes as well. Furthermore, the experiments showed that the code transformation proposed here can even improve code quality produced by compilers that can already handle cyclic programs.

We have presented the transformation for Esterel programs; however, as mentioned in the introduction, this transformation should also be applicable to other synchronous languages, such as Lustre. Lustre is also a synchronous language, but data-flow oriented, as opposed to the control-oriented nature of Esterel. To our knowledge, none of the compilers available for Lustre can handle cyclic programs, even though valid cyclic programs (such as the Token Ring Arbiter) can be expressed in the language. Hence in the case of Lustre, applying the source-level transformation proposed here is not only a question of efficiency, but a question of translatability in the first place.

Regarding future work, the transformation algorithm spells out only how to handle cycles carried by pure signals. We have presented an example for removing a cycle involving a valued signal, but this still has to be generalized.

There are also numerous optimizations possible, some of which presented in Section 5, which we plan to implement and evaluate; in particular, we are interested in the extent to which these optimizations might be helpful for Esterel programs in general, not just as a post-processing step to the transformation proposed here.

Finally, as we have observed earlier, the concept of constructiveness is a fundamental building block for the transformation presented here; constructiveness allows us to ultimately break a cycle by replacing the occurrence of a self-dependent signal in a replacement expression for that signal by an arbitrary value (`true` or `false`). However, if we would like to determine in the first place whether a program is constructive or not, the transformation proposed here might be employed to accelerate this analysis; by replacing signal occurrences by expressions as computed by the algorithm (including possible self-references), one may replace a generally computationally expensive iterative procedure, which is a classical approach to analyze constructiveness, by a more efficient analysis.

## Acknowledgments

We would like to thank Stephen Edwards for several fruitful discussions about different types of cyclic dependencies and for providing his CEC compiler as a solid foundation to build upon.

Klaus Schneider was the first to note that this transformation might also be helpful for constructiveness analysis, as after the transformation there would be no need to perform fix-point iterations.

We also thank Xin Li for conducting the hardware synthesis experiments.

Finally, we thank the reviewers for their detailed and constructive comments.



## References

- [1] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.
- [2] Gerard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999.
- [3] Gerard Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.
- [4] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [5] Gerard Berry and the Esterel Team. *The Esterel v5\_91 System Manual*. INRIA, June 2000.
- [6] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications: International Conference Proceedings*, volume 735 of *Lecture Notes in Computer Science*. Springer, June 1993.
- [7] CEC: The Columbia Esterel Compiler. <http://www1.cs.columbia.edu/~sedwards/cec/>.
- [8] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In Florence Maraninchi, Alain Girault, and Éric Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, July 2002.
- [9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [10] Stephen A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2), February 2002.
- [11] Stephen A. Edwards. Making Cyclic Circuits Acyclic. In *Proceedings of the 40th conference on Design automation*, June 2003.
- [12] Stephen A. Edwards and Edward A. Lee. The Semantics and Execution of a Synchronous Block-Diagram Language. In *Science of Computer Programming*, volume 48. Elsevier, July 2003.
- [13] Esterel web. <http://www-sop.inria.fr/esterel.org/>.

- [14] The GNU compiler collection. <http://gcc.gnu.org/>.
- [15] Nicolas Halbwichs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [16] Jan Lukoschus, Reinhard von Hanxleden, and Stephen Edwards. Efficient compilation of cyclic synchronous programs. Technischer Bericht 0402, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, April 2004.
- [17] Sharad Malik. Analysis of cyclic combinatorial circuits. *IEEE Transactions on Computer-Aided Design*, 13(7):950–956, July 1994.
- [18] Paritosh Pandya. The saga of synchronous bus arbiter: On model checking quantitative timing properties of synchronous programs. In Florence Maraninchi, Alain Girault, and Eric Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.
- [19] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, University of California at Berkeley, May 1992.
- [20] Thomas R. Shiple, Gerard Berry, and Herve Toutati. Constructive Analysis of Cyclic Circuits. In *Proc. International Design and Test Conference ITDC 98, Paris, France*, March 1996.
- [21] Olivier Tardieu. Goto and Concurrency - Introducing Safe Jumps in Esterel. In *Proceedings of Synchronous Languages, Applications, and Programming, Barcelona, Spain*, March 2004.